EPICURE: a partitioning and codesign framework for reconfigurable computing

Jean Philippe Diguet[a], Guy Gogniat[a,*], Jean Luc Philippe[a], Yannick Le Moullec[b], Sébastien Bilavarn[c], Christian Gamrat[d], Karim Ben Chehida[e], Michel Auguin[e], Xavier Fornari[f], Anne Marie Fouillart[g], Philippe Kajfasz[g]

[a]Lester Laboratory, CNRS FRE2734/UBS University, Research Center, BP92116-56361 Lorient, France
[b]CISS, Fr. Bajers 7, DK-9220 Aalborg, Denmark
[c]EPFL/STI/ITS/LTS2, ELD 241, Station 11, CH-1015 Lausanne, Switzerland
[d]DRT/DTSI/SARC/LCEI CEA-Saclay, F-91191 Gif sur Yvette, FRANCE
[e]I3S Laboratory, CNRS/UNSA University, les algorithms, route des Lucioles, BP121-06903 Sophia-Antipolis, France
[f]Esterel Technologies, Twins 1, 679 Av Julien Lefèbvre, 06270 Villeneuve Loubet, France
[g]THALES, Land & Joint Systems, EDS/DHD, 146, Boulevard de Valmy, 92704 Colombes cedex, France

*** corresponding author:** guy.gogniat@univ-ubs.fr Fax number (+33) 2 97 87 45 27

Abstract

This paper presents a new design methodology able to bridge the gap between an abstract specification and a heterogeneous reconfigurable architecture. The EPICURE contribution is the result of a joint study on abstraction/refinement methods and a smart reconfigurable architecture within the formal Esterel design tools suite. The original points of this work are: i) a generic HW/SW interface model, ii) a specification methodology that handles the control, and includes efficient verification and HW/SW synthesis capabilities, iii) a method for parallelism exploration based on abstract resources/performance estimation expressed in terms of area/delay tradeoffs, iv) a HW/SW partitioning approach that refines the specification into explicit HW configurations and the associated SW control. The EPICURE framework shows how a cooperation of complementary methodologies and CAD tools associated with a relevant architecture can significantly improve the designer productivity, especially in the context of reconfigurable architectures.

Keywords: Smart HW/SW interface; Formal programming model; Design space exploration; Reconfigurable computing, Hardware/Software partitioning; Parallelism exhibition and functional estimation; design productivity

1. Introduction

Complex applications combine control and data processing that are generally tightly coupled. The design of an efficient architecture corresponding to a correct implementation of the application is a more and more tedious work. This design complexity is due for example to the HW/SW separation realized very soon in the design cycle, generally at the specification step, and which lasts over numerous refinement steps, until cosimulation. Approaches based on formal models can be an attractive way to deal with specification validation but there is still a lack of methodologies able to derive straightly from a formal specification an HW/SW architecture that realizes the specification. One major difficulty results from the architecture itself that combines different components, each one controlled in a specific manner that is not related with the control semantic in the formal model.

System-level design space exploration and platform-based design have been addressed by a certain number of research projects. Among them, [9, 10] and [11] are the most relevant approaches compared to our work. In [9] definitions and a methodology for platform-based design are presented. The main goal of this methodology is to improve the design re-usability and regularity in order to cope with the increasing development costs. However, they do not consider design space exploration which becomes mandatory due to the huge complexity of the design space. In SPADE [10] a modeling and simulation environment provides a design space exploration at several abstraction levels. The methodology consists in mapping applications described with Kahn Process Networks onto an architecture model which is roughly and manually defined by the designer in a first approach. Both the application and architecture models are simulated. The performance of the system is provided by the simulation results and is then manually analyzed by the designer to perform HW/SW partitioning and iteratively refine the architecture. This approach is very interesting, however many steps of the design flow are still performed manually. In [11], the exploration tool PLATUNE is presented which allows the exploration of the configuration space of parameterized SoCs. The method enables the identification of all Pareto-optimal configurations in terms of execution time and power. This is done by considering the dependencies between the SoC parameters such as cache size and associativity, bus width, buffer sizes etc. However their approach does not target automatic HW/SW partitioning.

Compare to previous efforts, the EPICURE project main contribution is to reduce the gap between a formal specification model and the architectural specification through automatic design space exploration and HW/SW partitioning steps. To achieve such a goal, we first consider a common view of the software and the hardware in the architecture in order to mitigate the difficulty to perform refinements on the formal specification. The common view of hardware and software is achieved in our approach by an intelligent interface (ICURE) between the software unit and a dynamically reconfigurable unit. Based on this abstract homogeneous architectural model of the heterogeneous architecture we have developed a design methodology based on the visual synchronous SSM (Safe State Machine) model integrated in the Esterel Studio™ framework. The methodology consists of an abstraction tool that estimates implementation characteristics from a C-level description of a task and a partitioning refinement tool that performs the dynamic allocation and the scheduling of the tasks according to the available resources in the architecture. Thus our approach alleviates drastically design efforts by bridging the gap between the specification and the heterogeneous target architecture dealing with dynamic reconfiguration. Such an approach, combining automatic design space exploration and partitioning steps corresponds to an original work and aims to a better adoption of dynamic reconfigurable heterogeneous architecture within embedded systems.

The EPICURE project results from a three year fruitful project (2000/03) based on a tight cooperation between several research efforts in both academia and industry areas. One of the main objectives was to set and experiment a framework of cooperative CAD tools in the context of reconfigurable architectures. Thus in this paper, due to the limited space, we aim to explain the philosophy of the EPICURE design flow and will not deeply explain all the underlying algorithms and architecture details. In the following sections we will first introduce the reconfigurable hardware/software architecture under assumption. In a second part we will describe the general design flow and methodology. Finally, a third part will illustrate the EPICURE framework on two application examples to show how the EPICURE methodology can be used efficiently. These applications have been provided by industry partners in the context of the project.

2. "ICURE"- centric architecture

Reconfigurable computing is still a very active and promising field of research investigation. Since the first attempt triggered by the introduction of SRAM based FPGAs in the late 80's, computer architects have made numerous advances for the design and effective use of reconfigurable computers. Some of those first attempts focused on the system level and used off-the-shelf devices such as advanced FPGAs packed with microprocessors or complex function cores [1, 2]. Other work strictly addressed the design aspect and has led to the definition of new architectures and families that could be realized with today CMOS or future nano technologies perspectives. All of those contributions would certainly result into well engineered high performance devices but could fall short when faced with real world applications, due to the lack of easy programming tools.

Among the various possibilities of reconfigurable hardware (RHW) architectures, we chose to focus on the interface that is needed to link a standard CPU and a generic reconfigurable processing unit (RPU). Although this might seem a little conservative from an architectural point of view, it is obviously the right place to introduce the necessary circuitry in order to abstract the CPU/RPU communication channel. The high level of abstraction in the CPU/RPU interface will ease the standardization of communications and development of protocols needed for the HW/SW partitioning tools. This way, the methodology and tools developed in the EPICURE project is also made independent from any specific reconfigurable technology.

The interface has been called ICURE, an acronym for "Intelligent Control Unit for Reconfigurable Element". It acts as an hardware abstraction layer in between the CPU and the RPU (Fig. 1). Practically, ICURE main functions are to implement the link between the CPU/RPU and to manage the reconfiguration process in a smart way. The CPU could be any of RISC, VLIW, DSP, micro-controller, etc. and is connected to ICURE via a memory mapped register set through a standard address/data bus. The RPU is a generic reconfigurable "black box" with a number of bi-directional I/O ports and a reconfiguration link. Such a generic interface allows for the connection of any type of existing (FPGAs) or future reconfigurable device.

Reconfiguration of the RPU is driven by a dedicated management unit inside ICURE. Reconfiguration data are stored in a data structure called a context which encapsulates all the necessary bitstream, routing information and object code needed for proper configuration and execution of the RPU logic resources. Along with the configuration data, a context also contains the necessary information about how the CPU should use the RHW. In particular, the context indicates the number, types and sequences of parameters required for a successful hardware function call. The reconfiguration bitstream and object code are synthesized and compiled using the appropriate back-end tools for the RPU and CPU targets.

The main tasks that are performed by the ICURE interface are:

- Take care of the reconfiguration phase control (reconfiguration control). This is done through a simple automaton that implements the proper logic for sending reconfiguration data to the RPU. For example, when using standard "off the shelf" FPGA, this controller will take care of sending the bitstream to the FPGA.

- Handle hardware/software data routing and exchange. By programming a crossbar type resource, a convenient routing of signals is performed in between the RPU I/O ports (the hardware port) and the dual-port memory (the software port) for the selected context. In that respect, the ICURE interface allows for software data types (int, float, etc.) to hardware data types (signals) seamless conversion. The routing table is changed each time a new context is loaded.

- Handle data synchronization between resources. The ICURE interface includes a dual-port memory that allows sharing small amount of data (32 kbits) in an asynchronous way between the CPU and RPU. For larger data transfers, an external access is provided to the ICURE interface through a DMA port.

- Take care of the context control (ICURE Context Management). Contexts for a given application are stored in a dedicated memory usually during an initialization phase. Upon request from the CPU (through control registers), the context manager (ICM) will configure the required context. The ICM will first check if a context is already configured and not running and will eventually perform the configuration phase. Once a context configuration is initiated, the ICM extracts the three parts of the context data structure (HW data, I/O routing, SW code) and direct them to the corresponding ICURE resources (Reconfiguration control, Routing, Shared CPU memory).

All the ICURE interface features are made available to the controlling CPU through a set of API functions that allow for a seamless use of the reconfigurable resources. Using an hardware function context is basically a two step process: In a first configuration phase, the CPU asks for a given context by calling a load_context() function. This configuration phase is entirely and automatically performed by the ICURE interface. Upon its completion, the CPU can enter the execution phase and use the desired hardware function through a simple call mechanism. To some extent, this concept is similar to the Set/Execute model proposed in [3].

The introduction of the ICURE interface in between the CPU and RPU along with its dedicated API, allows for a standardized mechanism for the control and access of the hardware reconfigurable tasks.

The EPICURE exploration and partitioning framework and tools described hereafter will thus help in defining and switching contexts using the ICURE standardized approach using a two step approach:

1. Help to selecting the appropriate application functions and tasks for use in a context.

2. Introduce context switching at the right places inside the application control execution graph based on application and/or user constraints.

3. Specification/execution model

3.1 Specifications

The model is specified using the Safe State Machines (SSM) formalism developed by Esterel Technologies [5]. SSM is a visual synchronous model, integrated in Esterel Studio™. SSM is the commercial version of SyncCharts, an academic synchronous graphical model designed in the mid-nineties [15] as a graphical notation for the Esterel language [16, 17]. As such, SyncCharts was given mathematical semantics fully compatible with Esterel semantics. Like Esterel, SSMs are devoted to the programming of control-dominated software or hardware systems. Those systems are reactive, that is, they continuously react to the stimuli coming from their environment by sending back some other stimuli. They are purely input-driven: they react at a pace imposed by their environment. A reactive application usually performs both data-handling and control-handling. Esterel and SSMs are imperative languages especially designed to deal with control-handling. In particular, they support hierarchical automata, parallelism and preemption mechanisms at any level of nesting.

The key point here is that the underlying semantic guarantees an unambiguous interpretation of an Esterel Studio™ design. Furthermore, thanks to this semantic, one can produce “Correct-by-Construction” code from the specification.

3.2 The Esterel Studio approach

Esterel Studio is a design environment optimized for complex control-dominated IPs (such as DMAs, protocols, cache controllers, I/O subsystems, etc.). It is dedicated to the capture of formal design specifications using SSM, and enables the formal verification of properties very early in the design phase. Its use automates the production of synthesizable RTL code (VHDL and Verilog), both for prototyping and production purposes. Thanks to its underlying model, it is also possible to apply formal verification on models. The designer defines a property and the Esterel Studio verifier can check if that property holds. The property is itself expressed using the Esterel Studio graphical or textual formalisms: the designer keeps the same environment to develop his/her designs and their properties. If the property fails, the tool produces a scenario that can be reused for validation purposes with the simulator.

3.3 Specifying models for EPICURE

The EPICURE methodology proposes to describe the control of the application using the SSM, and to let the data processing part as calls to procedures within the design. The key point is to analyze such calls made during a computation cycle, in order to determine their final implementation. This analysis step results into a separation of two kinds of procedures: those implemented in hardware, and those remaining in software. Note that the main controller remains implemented as a software application. Currently there is no automatic way to define a split of the global controller into smaller ones.

To determine which procedure calls are made during a cycle, we use the OC (Object Code) intermediate format of the Esterel V5 compiler. This format describes the model as a flat automaton, where any internal communications in the original design have been resolved, and where parallelism and hierarchy disappear.

The format describes for each state, the transitions to the next states as a direct acyclic graph where leaves represent the next states. Along the paths of the graph, there are actions that are either typical procedure calls, or tests (input model tests, or internal data tests) that generate a branching to other graphs. From this graph, one can determine the calls that are performed, their order, and then analyze the effective cost of a transition. This is held by the partitioning process, as will be explained in Section 4.3. The OC format also contains all the procedures used by the model, which are used by the exploration tool to determine the procedures that have to be estimated and implemented in hardware or in software. Fig. 2 proposes a schematic description of the compilation flow using OC.

The SSM description is first compiled into some internal format, possibly OC. At this stage, one can already make a verification of activities. Applying verification during the design phase is an interesting feature, because it may help to find bugs in the specification at early stages, and brings more confidence to the system under design.

Then, one can go through the code generation process. In particular, it is possible to produce a C code for interactive simulation. If the verification phase exhibits some problems (a non-verified property), then the verifier generates scenarios leading to the counter examples. Those scenarios can be played interactively within the simulator.

Once the final design meets all the requirements, the final code can be generated. In our case, we generate the OC intermediate code that is then used by the partitioning tool. Since the Esterel compiler can read any of its internal formats and starts the compilation process from its corresponding level, we also use an OC code properly modified by the partitioning tool to perform the final compilation step.

4. Exploration and partitioning design flow

4.1 Overview

The EPICURE framework is based on the ICURE and API models and on the three-step design flow depicted in Fig. 3. The first step is to specify the application. This step relies on an original execution model that enables the designer to handle both control and data-flow parts of the specification in order to exhibit the reconfiguration and speed-up opportunities. The specification strategy is guided by the ability of the designer to make a decision about the control/data-flow boundaries. A simple way is provided to guide this choice according to the application analysis. We will see in Section 5 that these decisions can have a huge impact on the final results.

The specification is performed with Esterel Studio as presented in the previous section. At the lowest specification level, data-flow links are inserted through C procedure calls within the Esterel specification.

The resulting file is an OC format code. Then, the second step of the EPICURE flow is carried out by the Design Trotter framework (i.e. exploration and estimation) whose entry point is the list of C procedures from the OC code. The exploration and estimation step provides FPGA resources/clock cycles trade-off curves for all the functions included in the C procedures. Based on these results of HW and SW solutions, the third step consists in selecting a pertinent implementation regarding design constraints such as performance, area and power, but also considering the reconfiguration opportunities. The partitioning solution is then given in the OC format. In case of an HW implementation, the C procedure call is replaced by an HW execution directive that is sent to the ICURE interface to control the FPGA context loading and HW/SW data exchanges. Finally, the OC to C conversion is made by Esterel Studio.

4.2 Exploration and estimation

4.2.1 Introduction

This work is based on the following observations: Firstly, the trade-off between energy savings, area use and real-time constraints, which is mandatory for future embedded systems, imposes an optimized use of the silicon die. Thus the necessary improvements can only be reached through an implementation of massive computation parallelisms, a use of limited clock frequencies and the design of more or less dedicated hardware modules. This search of computation parallelism and dedicated hardware opportunities constitutes an important task of the design flow. This task is located before the complete definition of

the target architecture and can be described as a system-level architectural exploration. Secondly, if we now observe that CAD tools for co-synthesis and co-simulation have reached a reasonable degree of maturity, system-level exploration tools still remain at a research stage. However, fast estimations based on largely abstract and uncompleted specifications are strongly required at a system level [6]. Most of the current tools provide an exploration based on a fixed library of pre-characterized functions. Such libraries usually quantify a software implementation and one or two hardware solutions. Thus, the architectural exploration is bounded by the fixed granularity of functions and limited to an hardware/software partitioning. It appears that another stage has to be developed in order to really guide or parameterize some architectural choices. To cope with the architectural exploration process, we have implemented the exploration and estimation flow depicted in Fig. 4. The different steps of the framework are detailed in the following sections: Specification, Characterization, Exploration and Physical Mapping.

4.2.2 Contribution

The problem of exploration and estimation can be divided in two steps. The first one is the design space exploration problem that focuses on the definition of pertinent architectural solutions, and the second one is the estimation of the mapping results of a given architectural solution on a target FPGA (see [6] for details). The originality of our approach is to combine both aspects within a single framework.

Previous methods and tools depend on a designer experience to define an initial architecture that will be further refined by mean of design-space exploration. The first step of our method consists in exploring and exploiting the parallelism potential of an application. Those exploration results are reported using tradeoff curves, where a point corresponds to a potential architecture. This is a guidance tool for the designer that helps him to define the initial architectures. The finally selected architectures can then be refined by existing co-design methods or by the next steps of the EPICURE flow. Our contribution focuses on the three following items:

Exhibition and exploitation of parallelism. Parallelism is a key parameter in the design of digital systems. It has a direct impact on several performance parameters such as execution time, energy consumption and area. Therefore we seek to explore the parallelism potential of an application in terms of a) type 1 (data-transfer and data-processing), b) granularity level, and c) type 2 (spatial, temporal). In our work these aspects are addressed thanks to our graph-based internal representation and via the time constrained core of our estimator (used with several time constraints). This estimator rapidly provides dynamic exploration of an application by means of parallelism vs. delay tradeoff curves in which a point, as stated before, corresponds to a potential architecture. The designer has then access to a set of solutions from which he can extract the most promising ones regarding the constraints. These solutions can then be refined and mapped to technology dependent architectural models using the next steps of our flow [7,8] or existing co-design tools;

Target architecture definition. At the system level, we assume the target architecture is not defined yet. At that level the designer considers algorithmic operators (resources executing the operations found in the specification) since the goal is to guide the definition of the architecture. After the function set pruning, the designer can use the exploration tool at the architectural level where algorithmic operators are associated to pre-synthesized arithmetic operators.

Impact of specification choices. Since the core of the estimation framework is based on a fast scheduler, the designer can evaluate very rapidly the impact of algorithmic transformations. This feature is very important since it enables the exploration of several specifications for a given application.

4.2.3 Specification

In the EPICURE project, each C function issued from the system specification (i.e. C procedures) is transformed into a Hierarchical Control and Data Flow Graph (HCDFG) at the beginning of the estimation step. There are three types of elementary (i.e. non hierarchical) nodes in an HCDFG graph, the granularity of which depends on the granularity of the architectural model (Fig. 5):

- **A processing node** (P) represents arithmetic or logic operation, the granularity of the node depends on the architectural model: (ALU, MAC, +,-, etc.).
- A **memory node** represents a data-transfer (DT). The main node parameters are the transfer direction (read/write), the data format and the memory hierarchy level that can be fixed by the designer.
- A **conditional node** represents a test operation (if, case, loops, etc.)

Two main types of oriented edges are used to indicate scheduling constraints:

- A **control dependency** indicates an order between operations without data-transfer, for instance a test operation must be scheduled before the mutual exclusive branches. The control dependency edges can also be used to impose a given order between independent operations or graphs with the intention of promoting resource optimization (data-reuse for instance).

- A **data dependency** between two nodes A and B indicates that node A uses a data produced by node B.

Based on these edges and nodes, the intermediate representation can be built (Fig. 5):

- A **DFG** is a graph that contains only elementary memory and processing nodes. Namely it represents a sequence of non-conditional instructions of the original C code.
- A **CDFG** is a graph that represents a test or a loop pattern with their associated DFGs.
- A **HCDFG** is a graph that contains only elementary conditional nodes, HCDFGs and CDFGs.

To build an HCDFG from a C code, the decomposition principle is quite simple. The graph is traveled with a depth-first search algorithm. An HCDFG is created when a conditional node is found in the next hierarchy level. When no more conditional nodes are found, a DFG is built. In order to ease the estimation process, CDFG patterns have been defined to identify rapidly "loop, if, etc." specific nodes. Another important point is that the model covers the processing complexity of the complete application. Thus, the computation of array indexes (address computation), conditional tests and loop index evolution are represented with DFGs. An HCDFG example is given in Fig. 5.

4.2.4 Characterization

The aim of metrics computation is to guide the designer in its choices of implementation using the functions characteristics. A first category of metrics can be computed before scheduling. The **MoM** metric indicates the ratio of data-transfer (global data) compared to the processing operations (local or reused data), so it indicates the bandwidth requirements of the graph. The **CoM** metric provides the ratio of test operations over all other kind of operations. A graph presenting a high **CoM** does not appear as a good candidate for hardware acceleration.

The second category of metrics is based on the ASAP-ALAP dates computed before DFG scheduling. The gamma metric provides the average parallelism over the critical path. A high value indicates that an important speedup for a given graph can be found (thus energy savings capabilities). The **DRM** is the Data Reuse Metric, namely the ratio between data transfers and computations. Compared to the **MoM** metric, the Data Reuse Metric is more accurate but also more complex. All metrics are first computed for DFGs. Then, they are extended to all granularity levels by mean of combinations, which rules are based on the graph patterns (parallel, sequential, exclusion, loop). More details are given in [19], note also that the HCDFG representation allows an easy implementation of various other metrics.

4.2.5 Exploration

The exploration is based on two sub-steps that are (C)DFG scheduling and (C)DFG combinations.

4.2.5.1 Adaptive DFG and CDFG-loop scheduling

Simple DFG: The estimation and exploration of the resource parallelism starts with the scheduling of DFGs. The objective is to quickly produce graph schedules for various cycle budgets in order to produce the parallelism/cycle budget tradeoffs illustrated in Fig. 4. The range of delay varies from the minimum cycle budget (**ΔC**) equal to the critical path to the delay required for a complete sequential execution path (**ΔS**). We use a time-constrained list-scheduling heuristic where the number of resources of type i allocated is given by the lower bound: Number of operations (i)/Number of cycles. This lower bound is recomputed online during the scheduling execution with the remaining operations and cycles in order to avoid final over-allocations. However, the scheduling policy is a key choice for resource minimization; it depends on the DFG orientation namely the critical resource (DT or P). So the **DRM** metric is used to cope with the priority scheduling order: DT scheduling first, P scheduling first or simultaneous DT/P scheduling.

Loop CDFG: We consider loop unfolding when a given time constraint cannot be reached. The minimum unfolding factor n is computed with the maximum distance of inter-iteration dependency dmax [12]. The loop core is duplicated (n times) and the ASAP dates of the duplicated graphs are shifted according to the value of dmax. Then the resulting DFG is scheduled.

4.2.5.2 CDFG hierarchical combinations

The extension to all granularity levels is obtained by a combination of the tradeoff curves previously computed for the DFGs and CDFG-loops of the function. The estimation process of a complete function is hierarchical. First, the lowest levels (DFGs) are processed. Then, the CDFGs are estimated using the loop and control patterns described previously. Finally, combination rules are applied in order to estimate sequential and parallel executions of graphs (and allow estimating all hierarchy patterns). During this bottom-up approach, the combination order is guided by the criticity metric: when three or more elements have to be combined, the two most critical ones are combined first and so on. Moreover, we aim at producing very fast estimates to allow the exploration of large design-space. So, considering the great number of points in each tradeoff curve, we cannot apply an exhaustive search of the Pareto points. Instead, we use a technique based on the peculiar points (PP) of the CDFGs tradeoff curves. A peculiar point corresponds to a solution for which the cost in terms of resources decreases for a given time constraint. Note that the term resource is used for processing units, data transfers and local

memories as well. Combinations are based on four algorithms according to the combination patterns (parallel, sequential, exclusion, loop). For each case, the combination rules are applied for processing and memory units. Briefly theses rules follow the general following ideas:

- Sequential (HC)DFG: cycle budgets are added and maximum values are considered for resource estimation.
- Parallel (HC)DFG: maximum of cycle budgets are considered and resources are added.
- Exclusive (HC)DFG: the designer can select the worst case in terms of cycle budget and resource or select a finer solution based on branch probabilities issued from traces. This last possibility has been used in the framework of the EPICURE project.
- Loop (HC)DFG: Loop scheduling is only considered for the deeper nested loop, higher levels of nesting are executed sequentially.

Thus, the tradeoff curves are finally available from the DFG to the top (HC)DFG level representing the whole function. As a result we obtain the following results for each HCDFG hierarchy level and for all pertinent cycle budgets:

- The number of processing operators used (MUL, ALU, MAC) depending on the association operation/operator defined in the architecture model;
- The size of local memories based on the combination of local memory sizes required for each DFG during the scheduling step.
- The total memory size required.
- The number of simultaneous memory accesses.
- The unrolling factor.
- The total number of control states.

The background memory size is computed with array first declarations. The framework provides also an histogram that represents the ratio of data types (bytes, long, etc.) and data nature (constant, temporary variables, arrays, scalar variable).

4.2.6 Physical Mapping

From the complete set of previous estimations (Fig. 6-a.), a designer can make a more relevant choice among the proposed SW and HW solutions according to the metric values and tradeoff curves. Then, the abstract scheduling profiles are projected onto an implementation technology under architecture constraints using a technology description (Technology Architecture and Performance Model - TAPM) and architectural rules (Fig. 6-b.). Separate models are used for control, memory and datapath parts in a way to achieve more accuracy and define realistic solutions.

Fig. 6 illustrates the estimation of the mapping process. As stated in Section 4.2.5, the abstract scheduling profiles are characterized for a given cycle budget by the number of processing operators, the estimation of the memory size, the number of simultaneous memory accesses, the number of control states and the unrolling factor. We defined this way a RTL characterization from which we can derive accurate estimations of resource occupation vs. execution time for a given target technology (expected results of the physical mapping). Results are then provided in terms of logic cells (e.g. CLB, logic elements) and dedicated cells (e.g. embedded memories, DSP blocks) use for a given execution time of the algorithm. Information about the target technology such as operators/memories area and delay are thus needed and described in the TAPM file.

For better accuracy, a specific computation process is used for each unit of the architecture:

- **Memory**

A basic approach is applied to compute the area of the memory unit from the RTL characterization (number of simultaneous memory accesses, total memory size) and the TAPM (area and access times of memories). This computation process is capable of considering different types of implementation cells (logic or dedicated cells), thus to provide even more realistic estimation values. The area of the memory unit is then estimated using the following rule:

$$N_{CELL} = MS * W_{RAM} / N_{BITS/CELL}$$

where MS is the total memory size, W_{RAM} represents the bitwidth of the data to be stored and $N_{BITS/CELL}$ the number of memory bits implementable within a single cell. The same approach is extended to ROM memories.

The number of control signals for the memories is derived for the evaluation of the control cost:

$N_{CS} = N_{RAM} * N_{CS/RAM}$, where N_{RAM} is the number of memories (maximum of simultaneous RAM reads and RAM writes) and $N_{CS/RAM}$ is the number of control signals for each memory (typically, write enable and address signals left to the control unit in the current version).

- **Datapath**

The processing unit is estimated by adding the contribution of each processing operator which area is given by the TAPM file in terms of logic cells or DSP blocks. For example 3 eight bit adders requires 12 logic cells on a Virtex FPGA (slices in this case), as each adder requires 4 slices to be correctly mapped on this device. Here again the number of control signals is computed in order to derive the impact on the controller.

- **Control**

The area of the control unit is derived from the number of control states N_S and control signals N_{CS}. Control logic is supposed to be implemented using a ROM memory. The data bitwidth depends then on the number of states and control signals:

$$W_{CTL} = \log2(N_S) + N_{CS}$$

The total area of the control logic is estimated as the area of a $N_S * W_{CTL}$ ROM memory.

From the knowing of the memory, control and processing costs, we can derive a global area estimate by adding the contribution of each unit of the architecture. Then, the algorithm execution time is derived from the cycle budget and clock period defined during the scheduling process and this completes the physical mapping estimation process of the hardware functions.

Concerning software estimations, by now the processor models are too coarse to obtain accurate results. Instead, we consider the solutions that present a parallelism compatible with the target processor. Accurate estimations are obtained after profiling.

Results for the HW and SW solutions are then provided to the partitioning tool.

4.3 Partitioning

4.3.1 Introduction

The goal of partitioning in our methodology is to allocate and schedule the application tasks such that the whole execution time is minimized, with a constraint on the RPU resources. This partitioning step takes into account the dynamic reconfiguration capabilities of the RPU. Fig. 7 represents the main partitioning steps in the EPICURE design flow illustrated in Fig. 3. These steps require the following inputs:

- The estimations in terms of time/resources tradeoffs for each task of the application, resulting from the estimation step (*.estim*).
- A set of architectural parameters (*.param*) describing:
    - ICURE: memory size, access time and width, ICURE API descriptions.
    - the target RPU: reconfiguration time/LC, maximum resource size. The resources in the RPU are 'Cells' that can be grouped in two main categories: the Logic Cells (LCs) (functional elements for constructing logic) and the Dedicated Cells (DC) (e.g. Block-RAMs, Multipliers, DSP blocks).
    - the target buses: width, access time and throughput.
- The OC code generated by the ESTEREL compiler (*.OC*). The structure of an OC code consists typically of several lists of objects referenced by the automaton (lists of user defined types, procedures, actions…) and the explicit automaton itself. This automaton is described in the Mealy's formalism where actions (like conditional structures and external C procedure calls) are labeled on edges. For example, in Fig. 8, the action list 0 1 2 3 4 5 between the states <1> and <2> consists of a series of 6 external C procedure calls.

We first start by analyzing the OC code to split the control part from the data flow part using our *OcDataExtractor* tool depicted in Fig. 7. Then we build the Data Flow Graphs (DFGs) considering transitions or paths in the automaton. Note that DFG for partitioning are different to DFG previously introduced for estimation, thus a node in a DFG for partitioning is a procedure call while a (HC)DFG for estimation is a graph representation of the procedure. In addition to the estimations (*.estim*) and the architectural parameters (*.param*), these DFGs represent the inputs of our partitioning method based on a genetic algorithm (GA). After partitioning all the DFGs, we get a set of schedules including computations on the CPU and the RPU, data communications and reconfiguration operations. This information is integrated in the initial OC control structure to rebuild a semantically correct OC code. The GA also delivers the set of contexts intended to be run sequentially on the RPU (*.CTX*).

We developed the *OcDataExtractor* tool to translate the OC code into an internal representation. This is made by adding test nodes when meeting conditional structures (IF statements, tests on ESTEREL signal presence) and then building a set of DFGs over transitions or paths of the automaton focusing on the external C procedure calls. The control part consists of the state sequencing of the automaton considering the inserted test nodes, and the data flow part consists of a list of external C procedure calls over each automaton transition or path.

4.3.2 DFG Construction

In order to guarantee coherence in the data locality between DFGs, we assume that all the data (variables) are stored in the dual port internal memory of ICURE. The tasks of each DFG, respectively, read (and write back) data from (to) this memory are performed at the beginning (at the end) of the execution. These explicit data transfers greatly limit the hazard and simplify the data coherency management. In Fig. 8, two memory nodes are added to our internal representation.

4.3.2.1 DFG construction over transitions

In this case, the automaton is split into transitions (in the sequel, we denote by transition the connection between two state nodes or a state node and a test node), and a DFG is built over each one.

A simple list of call actions does not exhibit the parallelism potential that can take advantage of the architecture capabilities. Therefore, a data dependence analysis is applied to the input and output parameters of the procedures to derive a DFG representation from this list of call actions. These actions operate on variables according to the following syntax:

Action_number: call Procedure_number (variables passed by reference) (variables passed by value).

A procedure call (denoted task in the sequel) can start its execution when all its parent tasks and incoming edges (in the sequel, we denote by edge the connection between two tasks or a task and the memory) have completed their own executions. Edges between tasks are annotated with the size (in Bytes) of the transmitted variables: procedure P_4, in Fig. 8, passes by value the variable 1 (of type 0: u_int = 2 Bytes) to procedure P_6 and procedure P_5 passes by reference the variable 2 (of type 1: user_type = 64 Bytes) to P_6.

4.3.2.2 DFG construction over paths

Partitioning can be applied to each DFG to get a set of configurations of the hardware and software units according to each list of procedure calls. However, according to the granularity level used to develop the ESTEREL description of the application, the volume of computation provided on each transition may be too weak to exploit the architecture efficiently. Therefore, we consider the ability to operate on paths of the automaton rather than single transitions.

Using a profiling of the application, we set the probabilities of transition on each exiting branch of a test-node and we speculate just once considering the most likely branch (90% of probability for the example of the transition between test node $test_1$ and state node 6 in Fig. 9). Considering more than one test-node in a path is feasible only if the composition of probabilities remains high enough. DFGs encountered in the path are concatenated into a single DFG (Fig. 9) by extending the data dependence analysis to all the global DFG tasks.

In the case where there is no data dependence between the tasks of two contiguous DFGs of the same path, we add control dependence edges (dashed edges without data exchange) assuring the temporal dependencies of actions belonging to transitions between successive states (example of the control dependence edge between tasks P_6 and P_7 in Fig. 9). In the contrary case, we save a write/read operation compared to a DFG construction over a transition. Task P_6 in Fig. 9**,** no longer have to store its computation result in memory but send it directly to P_8.

Partitioning DFGs over a path including test-nodes allows the reconfiguration unit to be pre-configured according to the most likely branch.

Once the DFGs is built, our goal is to partition the different DFGs associated with transitions (or paths) of the automaton, minimizing the global execution time under a maximum RPU resource constraint. Our approach, based on a Genetic Algorithm detailed in [18], performs an offline, non pre-emptive and static scheduling of the DFG tasks.

4.3.3 HW/SW partitioning using a Genetic Algorithm

To take into account the dynamic reconfiguration capabilities of this kind of architectures, the partitioning problem can be split in three different steps:

- Spatial partitioning: the classical assignment of the DFG tasks to the architecture components.
- Temporal partitioning: an architecture dependent step that consists of the definition of the different configurations that will run sequentially on the RPU. Configuration sequencing virtually extends the application space. Complex algorithms that require more processing resources than available on a given RPU are split into separate configurations and executed sequentially.
- Scheduling: a scheduling that handles the reconfiguration times (the times for configuration switching) in addition to the execution and communication times.

4.3.3.1 The spatial partitioning

This partitioning step is performed by the solution encoding step of the GA. This allows the GA to process a design space exploration by generating different mappings of the tasks on the CPU and the RPU. Note that this mapping scheme is not a binary binding of each task to the CPU or to the RPU (as the one in [14]), but a binding to an implementation point (from the estimation phase).
SW and *HW* runtimes of each task are estimated as explained in Section 4.2. A zero-LCs implementation point of a task denotes a *SW* only implementation. The number of implementation points can differ for each task depending on the exploitation of the available parallelism within the task.
The encoding of any solution corresponds to the binding of each task to an implementation point. The area is evaluated as a set of 'Cells' LCs and DCs. As a consequence, the Area/Time tradeoff curves are multi-dimensional. The GA encoding method codes a chromosome *C* with an array of genes of length *N* where *N* is the number of tasks. Each gene *C(i)* is an integer representing a percentage. The maximum 100% value is associated with the most *LCs*-based expensive implementation of task τ_i. The implementation point selected is the nearest point to *C(i)* on the *LC* axis. If there is more than one implementation point representing the same *LC* number, we proceed with a projection of the implementation points on the area plan (Fig. 10) and compare the *DCs* on the next dimension picking also the nearest point to *C(i)* on that axis, and so on. All the solutions delivered by this encoding method are viable. This permits us to code the multidimensional coordinates with a single integer, and greatly facilitates the conception of mutation and crossover generation operators.

The RPU affected tasks must be gathered in contexts (or clusters) to finally achieve the solution evaluation. A solution is evaluated by its overall execution time. This requires defining a schedule of the tasks including reconfigurations and data transfers between tasks.

4.3.3.2 The Temporal partitioning and scheduling

The temporal *partitioning* and scheduling steps are considered jointly because of the important interaction existing between those two problems. These are solved by the individual evaluation step of our AG.
Then the goal is to gather the *HW* allocated tasks, for each DFG, into *n* different contexts with the following conditions that:
- The size of each context does not exceed the maximum RPU size.
- There is a precedence relation between the *n* contexts.
- The data to be passed to the next context are buffered in the ICURE memory.

The GA evaluation step is based on a clustering heuristic inspired by the COSYN method [4].
Let's assume that an encoding solution allocates only tasks P_3 and P_5 of the DFG example of Fig. 9 to a *SW* implementation (Fig. 11) and all the other tasks to *HW*. Given the architectural parameters *(.param*) and the size of the transferred variables (labels of the edges), preliminary communication times can easily be computed [18].

We first assign priority levels to the tasks, starting from the DFG' leaves. The priority level of a task is the longest path from the task to a leaf evaluated as computation and communication costs (the task priority is in parentheses Fig. 11–a). To reduce the schedule length, we need to decrease the length of the longest path by clustering tasks (grouping tasks in contexts) along it in order to reduce the communication costs

We pick the unclustered task τ_i ($T_{exec}(i)$, $S_{LC}(i)$,$S_{DC}(i)$) with the highest priority, where $T_{exec}(i)$ is the execution time and $S_{LC}(i)$ (respectively $S_{DC}(i)$) the number of LCs (respectively DCs) defined by the corresponding implementation point (for simplicity reasons, we set the execution time and resource utilization of each task to a constant given by the table of Fig. 11). This context building is iterated with the next unclustered task presenting the highest priority task until the available maximum resources of the RPU is reached (in terms of LCs and DCs). Then, a new cluster is created and the process is repeated until all the *HW* mapped tasks are assigned to clusters.
For a maximum RPU size of 400 LCs and 100 DCs of the example of Fig. 11, we build two RPU contexts of four tasks each.

With partial reconfiguration capabilities, the reconfiguration time depends on the quantity N_k of LCs needed for mapping the context *k* on the RPU. We evaluate the reconfiguration time of the context *k* by:

$T_{reconf}\ (k) = N_k.T_{reconf/LC}$

Were $T_{reconf/LC}$ is the reconfiguration time per LC. When the RPU only supports complete reconfiguration, $T_{reconf}\ (k)$ is set to the total reconfiguration time. Partitioning approaches like [13] only consider total reconfiguration of the RPU.

Partial and complete reconfigurations are supported in this methodology, but overlaps between computations and reconfigurations are not considered. This last feature transforms the temporal partitioning problem to a constrained placement one, which is difficult to solve if we do not consider any assumption on the size and shape of the configurations [14].

The reconfiguration time for the two contexts of Fig. 11 is:

$T_{reconf} = 4 * S_{LC} * T_{reconf/LC} = 10$ time units.

Once the contexts are defined, the algorithm updates the *intra-Context* and *inter-Contexts* communication times. *Intra-Context* communication times are set to zero and *inter-Contexts* communication times are computed considering the data writing (reading) to (from) the ICURE memory and the reconfiguration time of the next context.

Hence, after updating the communication times, the global execution time is computed with the ASAP execution time of each task. The maximum ASAP execution time among all the leaves is the cost of the solution. The cost of the solution proposed in Fig. 11 is the global execution time = 108 time units.

4.3.3.3 The other steps of the GA

The fitness (execution time) of every chromosome (solution) delivered by the GA is evaluated by allowing its ranking onto the current population. Afterwards, the GA selects (selection step) the parents that will reproduce in the next generation (generation step). Finally, a renewal step is necessary to integrate the new offspring into the current population.

- The selection step: The selection of solutions by the GA is performed by a *Tournament* technique. A number ($N_{parents}$) of tournaments are performed, each one opposes a given number of individuals randomly chosen in the current population to finally select the fittest to be one of the parents allowed to reproduce.

- The generation step: Mutation and crossover operators are used on the $N_{parents}$ selected individuals to generate the $N_{children}$ solutions representing the new offspring.

- The renewal step: After generation of the new offspring, the renewal of the population is performed according to the *elitism* principle. *Clones* are not allowed in our renewal procedure. When a number of generations N_{gen} has passed without improvements of the best individual, the GA stops and displays the best encountered solution.

4.3.4 OC updating after partitioning

After partitioning all the DFGs deduced from the external procedure calls in the explicit automaton generated from ESTEREL, we get a set of schedules including computations on the processor and on the reconfigurable unit, data communications and reconfiguration operations. This information is integrated in the initial OC control structure by the *OcRebuilder* tool (Fig. 7). The GA also delivers the set of contexts that will run sequentially on the RPU, for the *HW* synthesis tools.

In our approach, we consider that the control structure (automaton states sequencing and test nodes executions) is executed by the processor. The processor must also execute the tasks allocated to itself and must control the reconfigurable part through the ICURE interface. This is achieved by a back-annotation of the OC code considering an interrupt-based execution scheme, a generic scheduling function on the processor and the use of APIs implemented in ICURE for context management.

5. Results

Two benchmarks are considered to illustrate the EPICURE paradigm: a video supervision application (ICAM) and a part of JPEG 2000 decoder. The former illustrates that exploration and estimation and partitioning can be applied efficiently on dataflow dominated applications and the latter shows a way to cope with complexity in the case of control dominated applications. The EPICURE project has been performed within a national research program where both previous applications have been selected to demonstrate the efficiency of our methodology and the underlying architecture. No other applications were selected to be part of the research project, thus the results section will be limited to the ICAM and JPEG 2000 case studies.

5.1 ICAM

5.1.1 ICAM description

The first benchmark used to evaluate the result quality of our approach is a video supervision application. This application is a part of the ICAM (Intelligent CAMera) project involving the LCEI lab of the CEA in collaboration with Thales, Atmel, Philips, WV, DC, Siemens, Alstom etc. that aims to develop an intelligent, cheap, and miniaturized camera with a high embedded computation power.

As depicted in Fig. 12, the video supervision application performs the motion detection and the object labeling given a fix background image. This application has a real time constraint of 40 ms per image.

5.1.2 ICAM Results

Starting from a C ANSI description non-optimized for a particular platform, we notice that this application is a set of sequential and iterative data flow treatments limiting de facto the efficiency of a parallel coarse grain architecture. So we introduced a level of parallelism by pipelining a part of the application (Background subtraction and filtering). The whole

benchmark is a SSM/C specification. Considering a coarse grain decomposition of the application, we set the number of external C procedures to 16.

The Esterel compiler generates an ***oc*** file with a 16 state automaton. The *OCDataExtractor* tool applied to this automaton builds a set of 14 DFGs considering transitions and 10 considering paths. These DFGs contain a maximum of 5 procedure calls per transition and 16 per path (we notice here that the path with the total 16 procedure calls of the application corresponds to the nominal path).

These 16 procedures are characterized upon our three metrics to guide the implementation choices as shown in Tab.1. The first step of the estimation has consisted in profiling the code with a set of video sequences. In the second step a selection of interesting functions (i.e., those with exploration potential) has been performed. This selection has been possible thanks to the characterization step, of which the results are shown in Tab.1. The third step was the intra-function exploration and estimation. This step is performed to explore the potential parallelism of the functions and generates resource vs. time constraints (abstract cycles) tradeoff curves. Then a new selection step has been performed in order to select the most promising solutions. To do so, we have used a speed-up vs. resource extra-cost criterion. Finally the selected solutions have been processed by the last step of the estimation flow: the physical mapping. In this case study, we have used the Xilinx V400EPQ2 with a reconfiguration time per LC of 5µs. Estimated implementation characteristics of functions are deduced from this physical mapping and provide execution time, LC# and DC# estimations (like the six implementation points of the convolution/histogram function in Tab.2). Given these values and execution time estimations on the ARM922 processor, we partition the DFGs on each transition and each path. The execution time of the genetic algorithm over all the possible paths is about 10 minutes with an initial population size of 300 individuals. Some HW projection results are presented in Tab.2. Finally, the exploration and estimation results have been transmitted to the partitioning step.

A partitioning result example over a transition is illustrated in Fig. 13. This Gantt chart shows the scheduling on the RPU, the CPU and the ICURE memory occupation highlighting the tasks execution time, communication time and reconfiguration time.

By summing the execution times of the transitions within the nominal path, we reach a total execution time of 36.1 ms as shown in Fig. 14 (*OC_Transition*) and 32.4 ms considering the nominal dataflow path (*OC_Path*). We compare these results with the partitioning of the entire application (using our method) on an ASIC equivalent technology (*HW_Only*) (leading to a resource utilization of 5974 LC + 282 BlockRAM equivalent to 71688 gates + 1128 kb dual port synchronous memory) as a *Single_DFG* (without adding control by pipelining the application) and the execution time of the *SW_only* implementation.

Considering the nominal path in the partitioning leads to bigger DFGs than those extracted on single transitions and therefore larger configuration contexts. So we find two contexts for *OC_Path* on a Virtex XCV2000E (43200 LCs and 150 BlockRAM) with a maximum utilization of 3596 CLs and 150 BlockRAM per context, and 10 contexts for *OC_Transition* on a smaller Virtex XCV1000E (27648 LCs and 96 BlockRAM) with a maximum utilization of 2082 CLs and 96 BlockRAM per context. As mentioned in Section 4.3.1.2*,* the DFG construction over the automaton paths allows saving a significant data exchange time between the processing elements and the ICURE memory. That can be seen on Fig. 14, considering a one DFG construction over the nominal path (*OC_Path*) gives an improvement of 4 ms compared to the concatenation of 12 DFGs (in *OC_Transition*).

The *HW_Only* design uses nearly twice the amount of resources used by the *OC_Path* version and three times the resources of the *OC_transition* one with indeed a significant improvement of the overall execution time but loosing the potential flexibility.

The ICAM benchmark has proven the ability of the EPICURE paradigm to efficiently explore coarse grain applications. We have also tested the behavior of our methodology when considering the JPEG 2000 Entropic decoder, which is a fine grain, control dominated application.

5.2 JPEG 2000

5.2.1 JPEG 2000 description

JPEG 2000 is a new image coding standard for different types of still images (bi-level, gray-level, color, multi-component) with different characteristics (natural images, scientific, medical, remote sensing imagery, text, rendered graphics, etc.) allowing different imaging models (client/server, real-time transmission, image library archival, limited buffer and bandwidth resources, etc.) preferably within a unified system.

In addition to the basic compression functionality, numerous other features are provided, including:

- Progressive recovery of an image by fidelity or resolution,
- Region of interest coding, whereby different parts of an image can be coded with different fidelity,
- Random access to particular regions of an image without needing to decode the entire code stream,
- A flexible file format with provisions for specifying opacity information and image sequences,

- Good error resilience.

Taking also into account wireless constraints, JPEG 2000 is well adapted to real-time embedded systems such as smart phones or digital video systems. It provides low bit-rate operation with rate-distortion optimization and subjective image quality performance superior to existing standards.

In the study, we focused on the decoder part. The Fig. 15 gives a description of the decoder process. Bytes are extracted from the JPEG 2000 codestream, then the parser read the data and, as soon as an entity (marker, packet header data, code block data) is ready to be decoded, the decoder processes it. Two types of data exist:

- Marker: initialize image information (size, size of the code block, type of wavelet transform, rate…)
- Packet & Packet header : input of the entropic decoder

When the parser is processed, the code block is decoded (entropic decoder) and two processing stages are executed (Fig. 16):

- Coefficient bit modeling which re-order and reconstruct the data
- Arithmetic decoder which takes bytes and contexts and generates bits to reconstruct wavelet coefficients

Finally, the process ends with, successively, the inverse quantization; the inverse wavelet transform and the inverse color transform in order to build the image.

5.2.2 Results on the JPEG 2000 Entropic Decoder

We tried to model the entropic decoder of the JPEG 2000 standard with no a priori choices on granularity level. So we have considered a very fine grain specification of the entropic decoder of the JPEG 2000 standard in terms of external C procedures (we used 27 C procedures at the granularity of: logical shifts, clear, comparison …). The Esterel compiler generated an automaton with 17158 states.

The exploration and estimation step for this application is not very relevant as the procedures are very simple and mainly composed of a few arithmetic operations. They mainly correspond to basic DFGs where the exploration potential is very low. Thus the parallelism exploration was transferred to the partitioning step.

The *OcDataExtractor* tool applied to the ***oc*** generated file builds 44924 DFGs considering transitions and 44158 DFGs considering paths with a maximum of 1 procedure call per transition and 3 per path. If we apply it to a subpart of the Entropic decoder with 311 states, we get 4037 DFGs considering transitions and 3183 considering paths. This high number of DFGs greatly limits the possibility to apply partitioning on each of them.

By scanning methodically the different generated DFGs, we have found that there is a large redundancy in their structure and content. So, we added a redundancy check function to the *OC2Automaton* tool to allow it creating just the non-redundant DFGs for the partitioning step. This procedure is based on a comparison of a sequence of actions operating on a list of variables. With this kind of functions, we are sure to compare the same list of actions (C procedure calls, test nodes) operating on the same variables.

The results are given Fig. 17. We can notice that the number of DFGs is drastically reduced: 99.7% less, from 44924 DFGs to 107 for DFG construction on the transitions of the total Entropic decoder automaton. The genetic algorithm can be easily applied to these DFGs providing results for the whole application. However, due to the very fine grain specification used to describe the entropic decoder, these results have a limited interest in practice. Actually, the complexity is no longer on the processing part but on the control one (the automaton states sequencing and test nodes executions) that will be implemented on the CPU. Further investigations are necessary for partitioning the control on the CPU and RPU. The aim of this example was an illustration of the ability of the methodology to deal with large complexity.

These results show also that the designer can not, in the specification phase, really make abstraction from the underlying exploration and estimation and partitioning tools when making choices on the processing granularity. Despite the complexity reduction capability of our methodology, the partitioning results led us to reconsider the SSM/C specification and the external C procedures granularity.

6. Conclusion

As a first result it is important to stress that the EPICURE philosophy has lead to a fully automatic process from a set of tools that includes graph extraction from a mixed SSM/C specification, system and architectural exploration and estimation and hardware/software partitioning. Such a methodology is made possible by the ‘ICURE’ hardware abstraction layer in between the CPU and RPU. As an additional benefit this generic interface allows for the EPICURE methodology to become applicable regardless of the hardware architecture of either the CPU or the RPU.

The EPICURE framework is a proof of concept regarding a complete design flow that alleviates the tedious system level design task for run time reconfiguration computing by allowing a fast exploration of numerous potential solutions. Therefore the overall design time is tremendously reduced as shown with ICAM experience.

The EPICURE experience also opens some complementary research issues regarding the specification step. On the one hand, the exploration and estimation process can be performed at different levels of granularity; however the top level is fixed by the designer by means of C functions calls. On the other hand, we observe that the partitioning tool can reduce the DFG redundancy, which usually shows up that a coarser grain specification level could have been chosen. Thus, it appears that the designer choices still influence the final results. But this effect could be drastically reduced with an additional step in the methodology dedicated to the application specification issue. EPICURE already brings some solutions. The characterization step of the exploration and estimation flow provides metrics to measure the control influence and can guide the designer in his choice. Moreover a blurred separation between Control and Data-flow can be considered by adding some control extraction capabilities in the EPICURE flow. From a general point of view, the combination of specification/design space exploration is opens an interesting research field with real productivity gain perspectives. We consider that one of the main JPEG 2000 results is the highlighting of this relevant issue.

The EPICURE experience could not be entirely condensed in a single paper. Thus, a hardware implementation of the "ICURE" interface is currently under progress, new architectural models have been added in the exploration and estimation flow to extend the potential system targeted. And power consumption has been considered so as to promote energy efficiency through dynamic HW/SW partitioning.

7. Acknowledgment

This project was funded in part by a grant from the French ministry of research under the RNTL framework.

8. References

[1] http://www.altera.com/products/devices/excalibur,

[2] http://www.xilinx.com/products

[3] Sima M., Vassiliadis S., Cotofana S., Van Eijnhoven J., Vissers K, A Taxonomy of Custom Computing Machines, 1st *PROGRESS workshop*, Utrecht, The Netherlands, October 13, 2000.

[4] Dave B.P., Lakshminarayana G., Jha. N. Cosyn: HW/SW co-synthesis of embedded systems, DAC, 1997.

[5] http://www.esterel-technologies.com

[6] Plantin P., and Stoy E., Aspects on system level design, 7th Int. Symposium on HW/SW Codesign, 1999,Rome,Italy

[7] Bilavarn S., Gogniat G., Philippe J-L., Fast Prototyping of Reconfigurable Architectures From a C Program, IEEE ISCAS, Bangkok, 2003.

[8] Le Moullec Y., Diguet J-Ph., Gourdeaux T., and Philippe J-L, Design Trotter : System-Level Dynamic Estimation Task a 1st step towards platform architecture selection, in Journal of embedded computing (JEC), Cambridge Int. Science Pub, issue 4, dec. 2005.

[9] Sangiovanni-Vincentelli A., Defining Platform-based Design, EEDesign of EETimes, Feb. 2002.

[10] Lieverse P., Stefanov T., Van Der Wolf T. and Deprettere Ed F., System Level Design with Spade: an M-JPEG Case Study, ICCAD, Nov. 2001.

[11] Givargis T., Vahid F. and Henkel J., System-level Exploration for Pareto-optimal Configurations in Parameterized Systems-on-a-chip, ICCAD, Nov., 2001.

[12] Le Moullec Y., Diguet J-Ph., and Philippe J-L, Design-Trotter: a Multimedia Embedded Systems Design Space Exploration Tools, IEEE Workshop on Multimedia Signal Processing, USA, dec., 2002.

[13] Chatha K.S. and Vemuri R., HW/SW Codesign for Dynamically Reconfigurable Architectures, 9th Int. Workshop on Field Programmable Logic and Applications (FPL'99), Glasgow, Springer Verlag, Sep., 1999.

[14] Mei B., Schaumont P. and Vernalde S., HW/SW Partitioning and Scheduling Algorithm for Dynamically Reconfigurable Embedded Systems, 11th ProRISC workshop on Circuits, Systems and Signal Processing, Veldhoven, Netherlands, Nov. 2000.

[15] André C., Representation and Analysis of Reactive Behaviors: A Synchronous Approach, Computational Engineering in Systems Applications (CESA), Lille (F), July 1996. Publisher: IEEE-SMC, pp 19–29.

[16] Boussinot F., De Simone R., The ESTEREL Language. Another Look at Real Time Programming, Proceedings of the IEEE, 79:1293–1304, 1991.

[17] Berry G., The foundations of Esterel. In G. Plotkin, C. Stirling, and M.Tofte, editors, Proof, Language, and Interaction: Essays in Honour of Robin Milner, MIT Press, 2000.

[18] Ben Chehida K. and Auguin M., Partitioning reactive Data Flow Applications On Dynamically reconfigurable, Systems, IFIP Conf. on Very Large Scale Integration of System-on-Chip VLSI-SOC'03, Germany 1 - 3 December 2003

[19] Le Moullec Y., Ben Amor N., Diguet J-Ph., Philippe J-L., Abid M., Multi-granularity Metrics For The Era Of Strongly Personalized SOCs, Design Automation & Test in Europe (DATE), March, Munich, 2003.

In the following pages all figure captions and all tables are defined

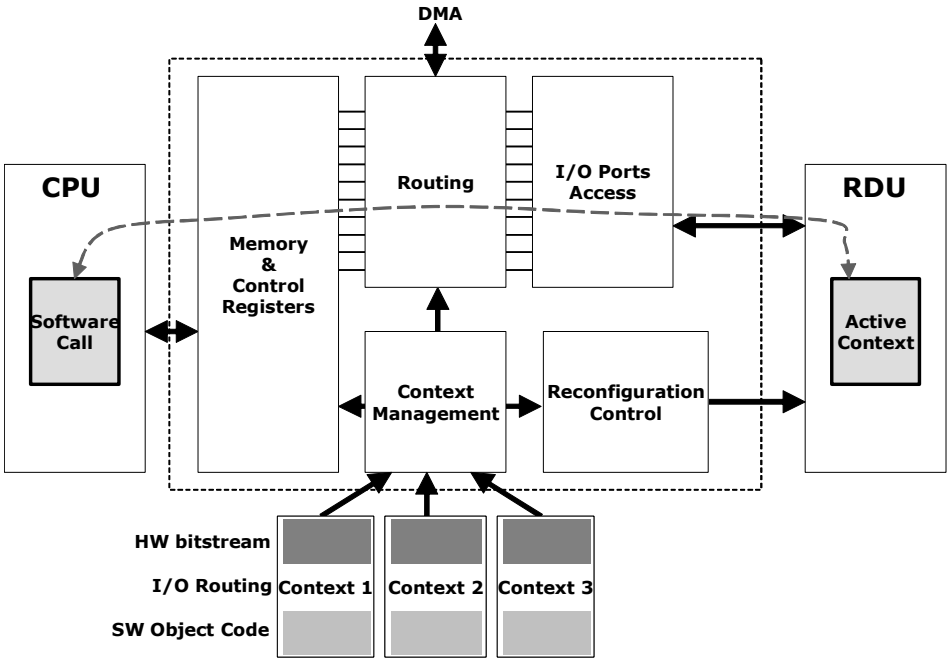


Fig. 1 - The "ICURE" interface enables communications between the CPU and the Reconfigurable Processing Unit

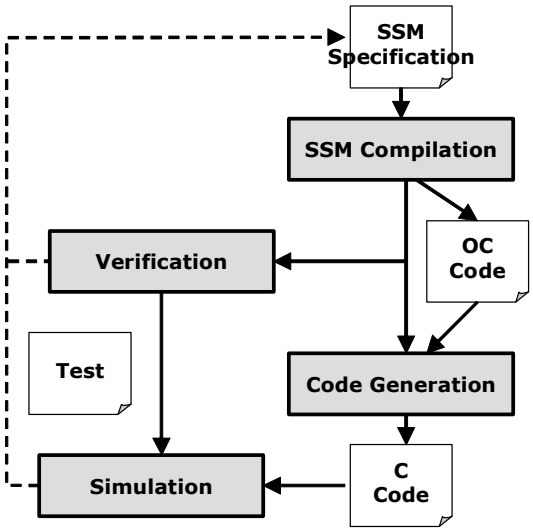


Fig. 2 - Compilation flow using OC

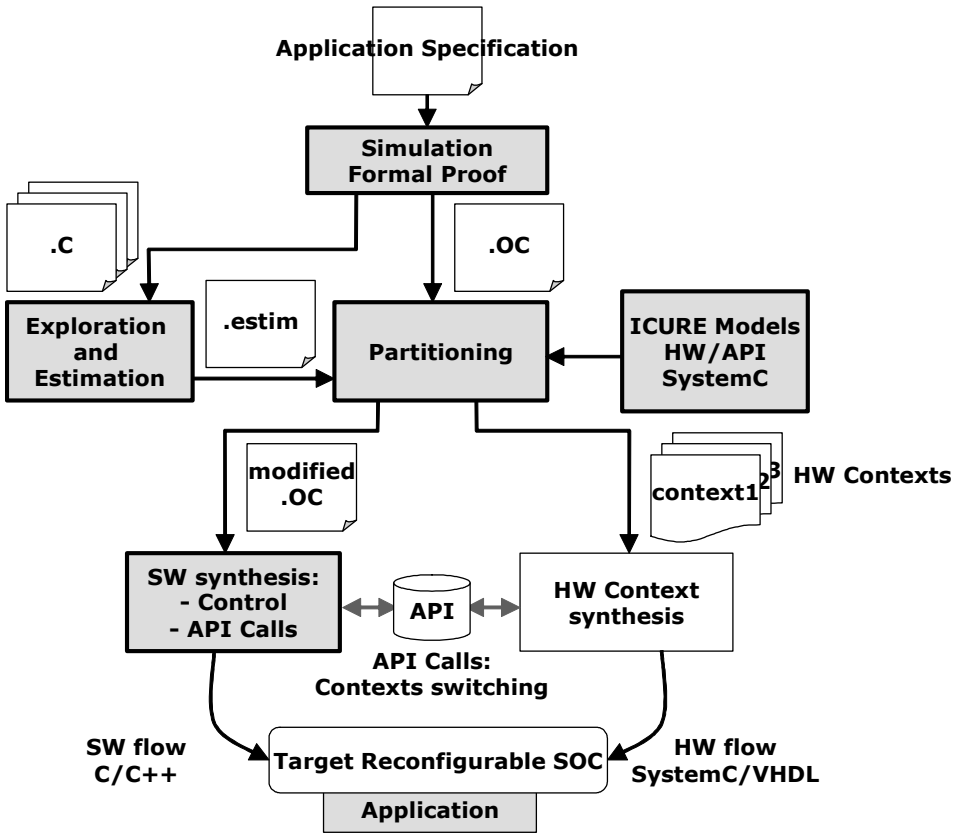


Fig.3 - EPICURE global design Flow

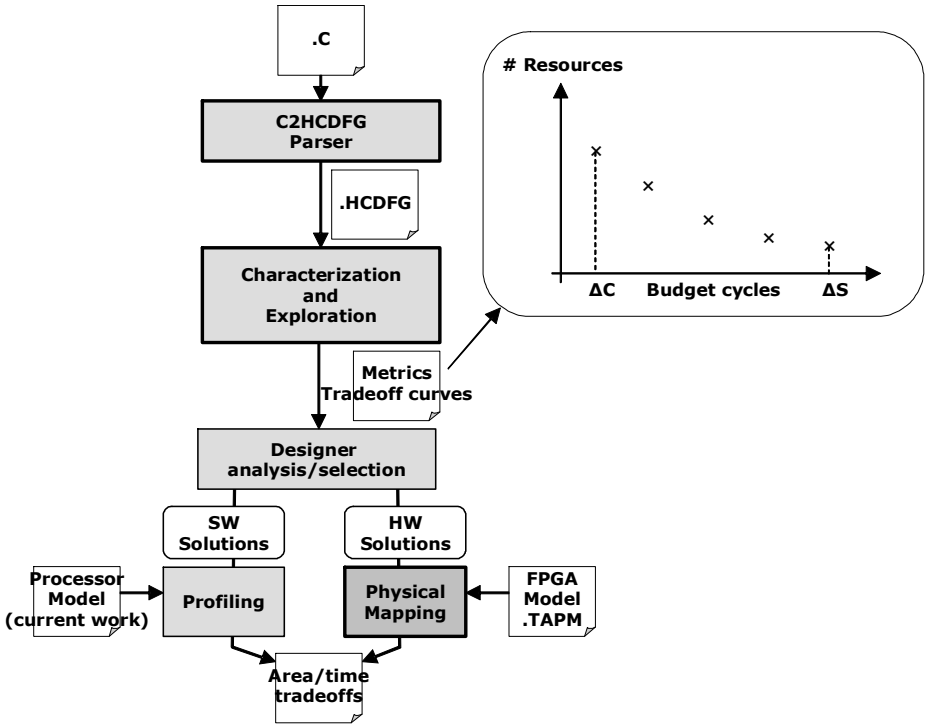


Fig. 4 - Exploration and estimation flow

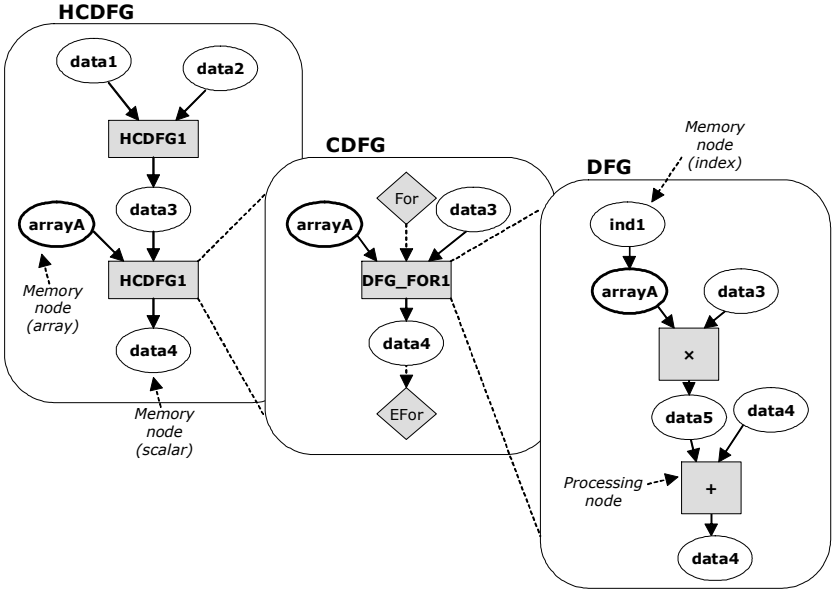
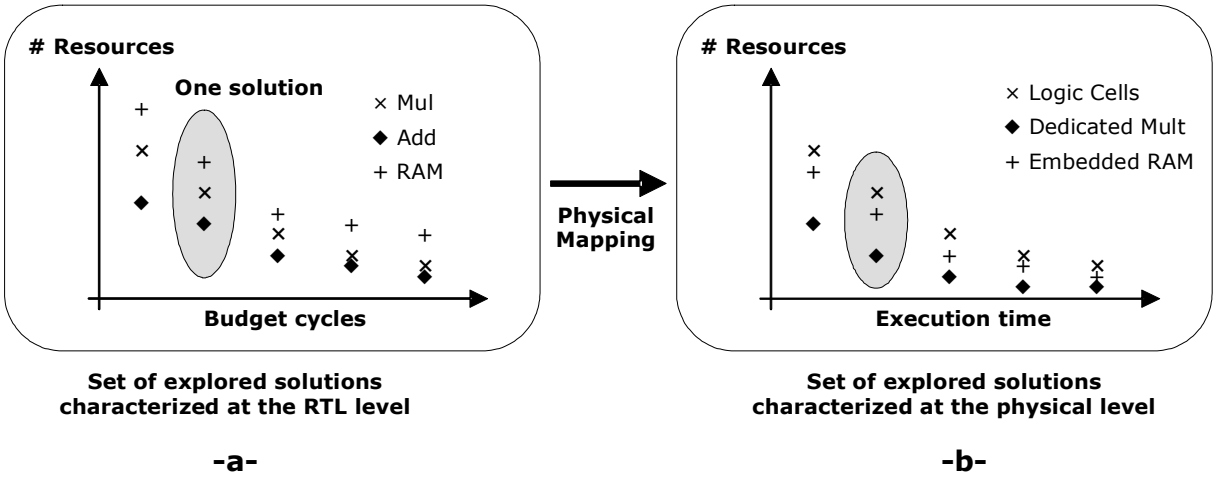


Fig. 5 - A HCDFG specification example



Set of explored solutions characterized at the RTL level

-a-

Set of explored solutions characterized at the physical level

-b-

Fig. 6 - HW tradeoff curves – parallelism exploration

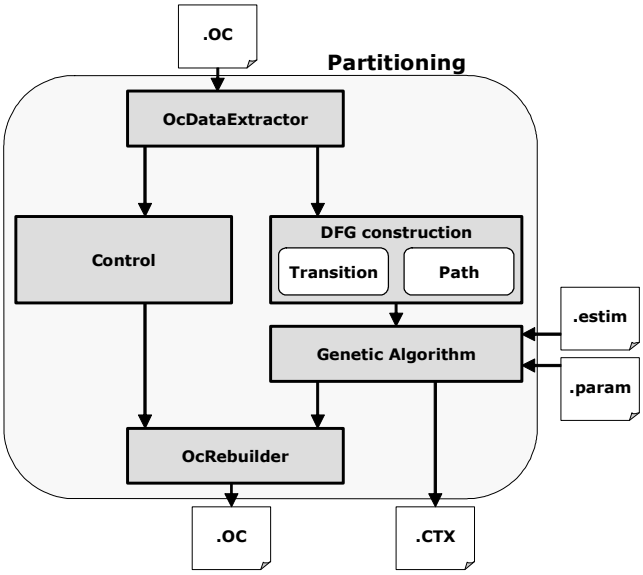


Fig 7. – The partitioning flow

OC Code

Types:
0: u_int
1: user_type

Procedures:
0: P_1(0, $1)()
1: P_2(0, $1)()
2: P_3(1)()
3: P_4(0, $1)()
4: P_5(1)()
5: P_6(1)(0)

Actions:
0: call: 0(1,3)()
1: call: 1(1,3)()
2: call: 2(2)()
3: call: 3(1,3)()
4: call: 4(2)()
5: call: 5(2)(1)
…

States:
1:
0 1 2 3 4 5 <2>
…

Type sizes (Bytes)

boolean:	**1**
integer:	**2**
string:	**1**
float:	**4**
double:	**8**
u_int:	**2**
User_type:	**64**

Our internal representation

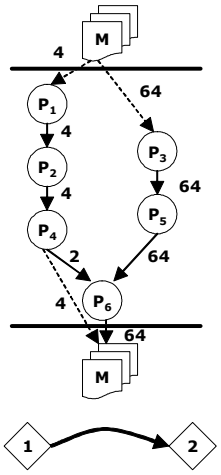


Fig 8. – Data dependency check and DFG construction

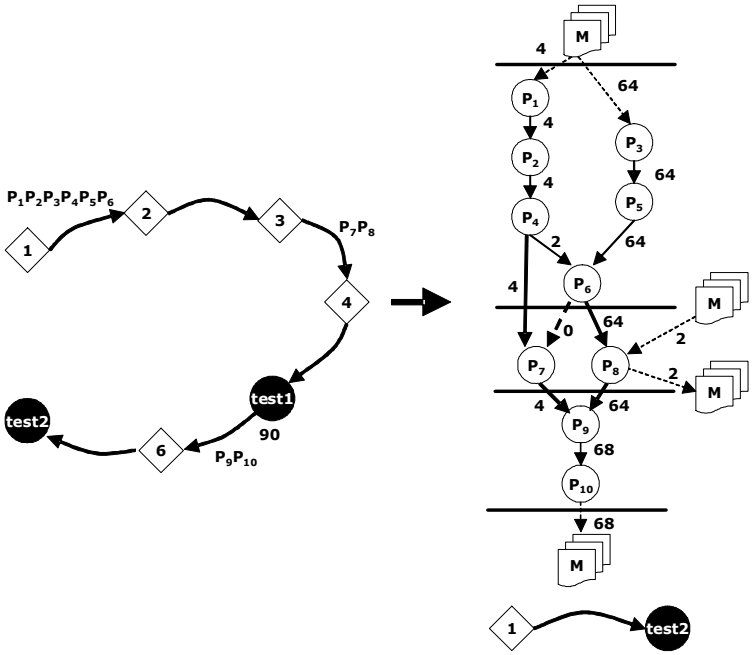


Fig. 9 – DFG construction over a path

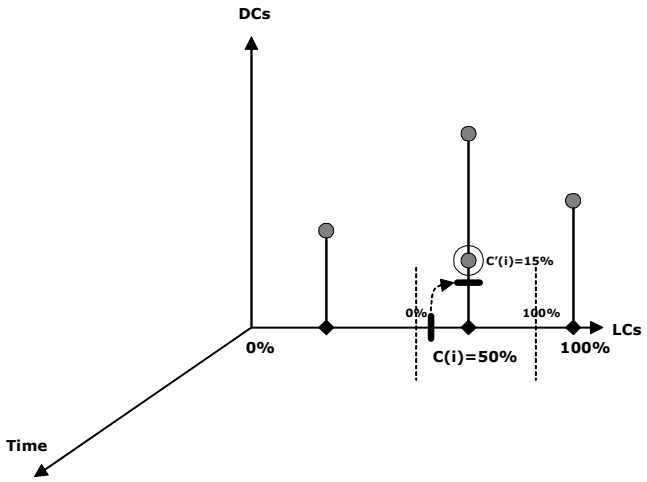
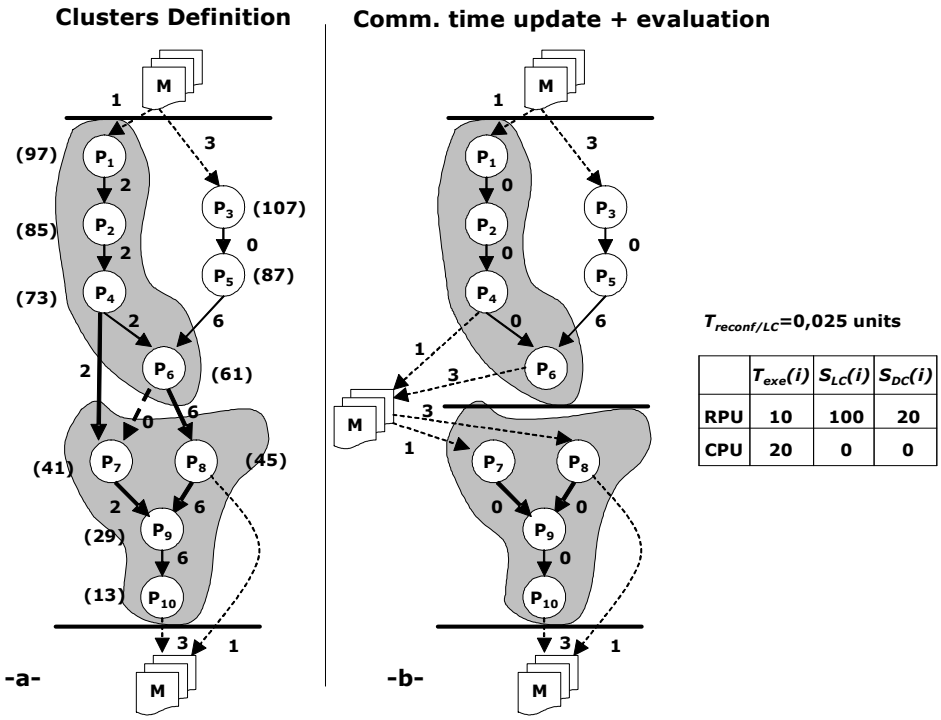


Fig. 10 – Implementation point selection



	$T_{exe}(i)$	$S_{LC}(i)$	$S_{DC}(i)$
RPU	10	100	20
CPU	20	0	0

Fig. 11 – Our clustering heuristic

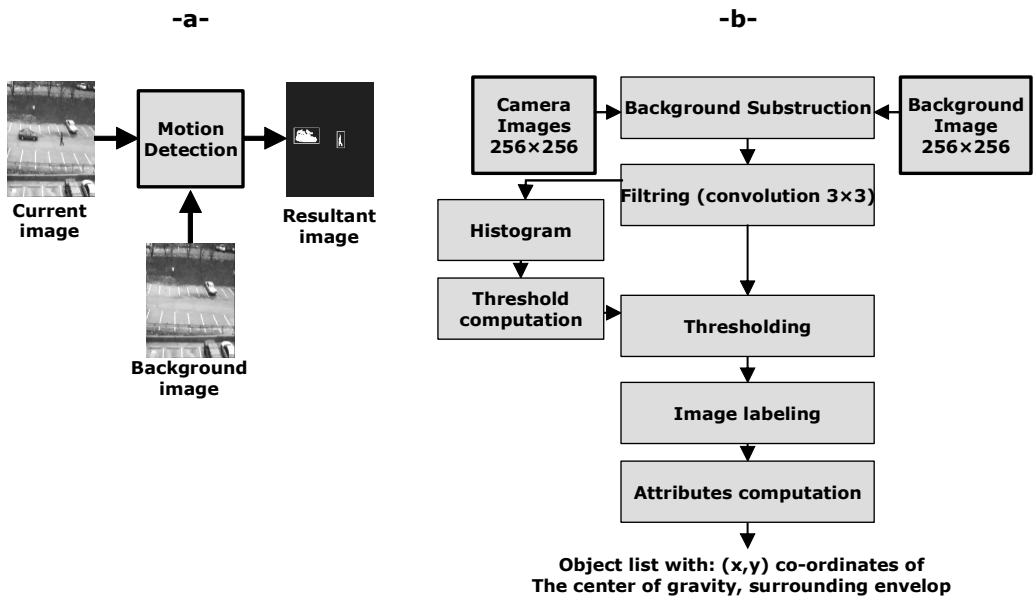


Fig. 12 – The ICAM video supervision application

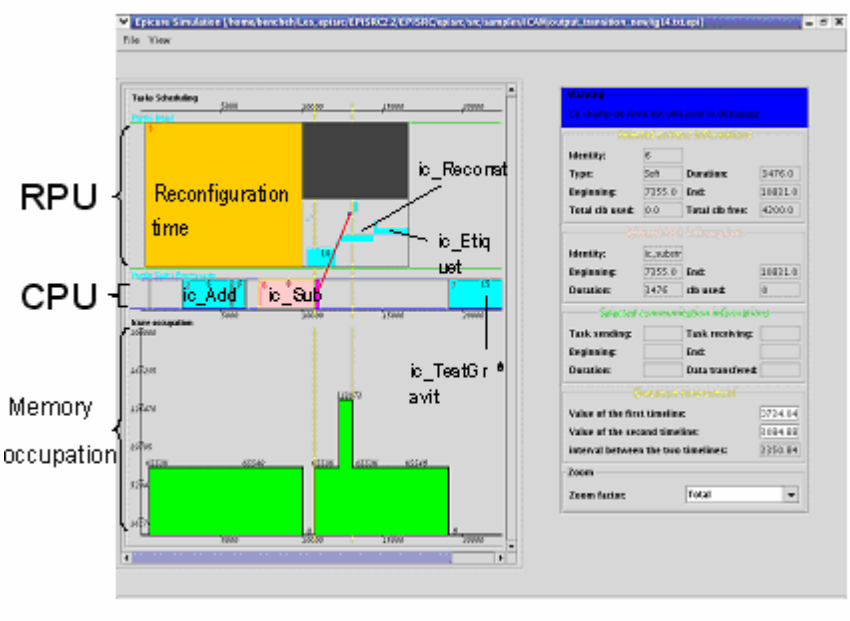


Fig. 13 – A partitioning solution over a transition

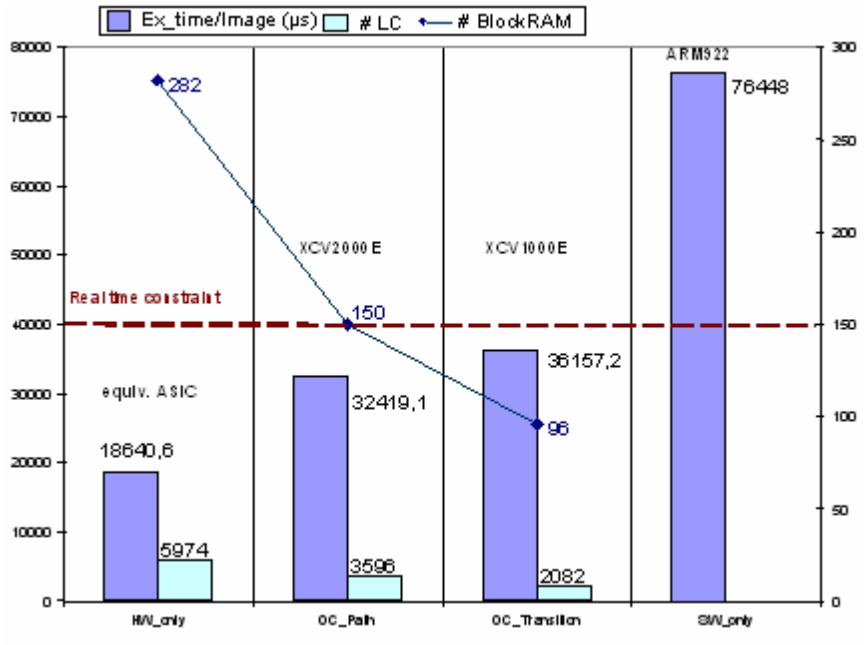


Fig. 14 – Execution Time/Image & resource use

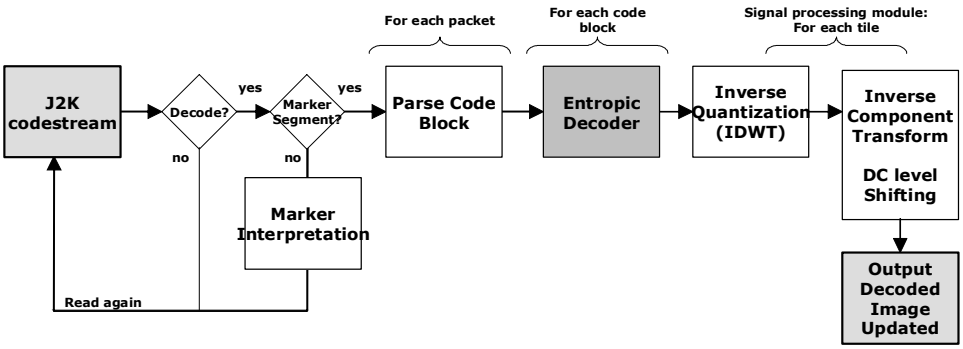


Fig. 15 - Decoder synopsis

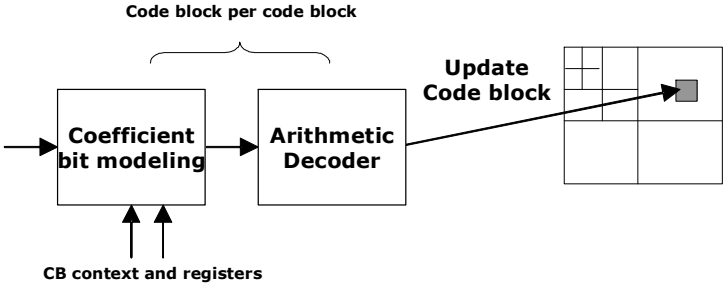


Fig. 16 - Entropic decoder

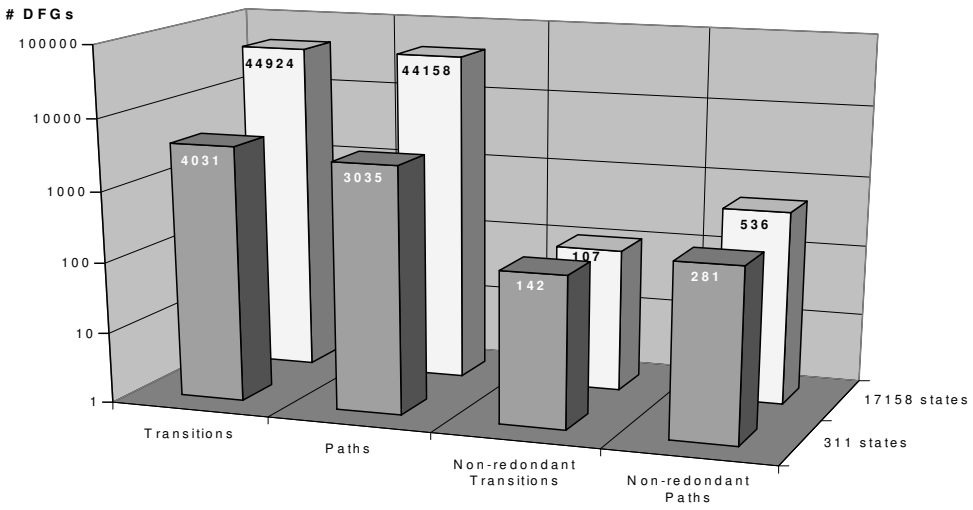


Fig. 17 - DFG redundancy reduction

	Gamma	MOM [0,1]	COM [0,1]
Ic_testGravity	43,88	0,78	0,22
Ic_labeling	10,31	0,74	0,07
Ic_changeBackground	5,62	0,76	0,03
Ic_reconstDilat	4,75	0,65	0,32
Ic_dilatBin	4,69	0,70	0,02
Ic_histoThreshold	4,00		0,29
Ic_envelop	3,91	0,66	0,13
Ic_absolute	2,60	0,71	0,08
Ic_thresholdAdapt	2,20	0,75	0,08
Ic_convolveTabHisto			0,03
Ic_div	1,25	0,73	0,00
Ic_getHistogram	1,22	0,75	0,00
Ic_setValue	1,14		0,00
		0,75	0,00
Ic_sub	1,11	0,75	0,00
Ic_erodBin	1,10	0,73	0,01

Tab. 1 - ICAM characterization step results

N°	Time (ns)	States #	LC #	DC #
1	20191042	4384	868	357
2	46196075	4415	428	191
3	82309179	857	340	56
4	153223136	614	296	36
5	194375278	618	296	32
6	266391525	625	296	28

Ex_time/Image (µs)

Tab.2 - Projection Results